
**Università di Pisa**

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

# Implementazione del sistema di protezione contro Meltdown nel nucleo didattico

Candidato:
**Riccardo Sagramoni**
**Matricola 565472**

Relatore:
**Ing. Giuseppe Lettieri**

**Anno Accademico 2019-2020**

# Indice

# Capitolo 1

# Introduzione

# Capitolo 2

# Meltdown e il sistema di protezione KAISER

La sicurezza dei sistemi informatici attuali si fonda sull'isolamento della memoria, ad esempio marcando come privilegiati gli indirizzi di memoria kernel e bloccando eventuali accessi da parte di programmi utente [9]. **Meltdown** è un tipo di attacco informatico che sfrutta un effetto collaterale dell'esecuzione fuori ordine nei processori moderni per leggere locazioni di memoria scelte in maniera arbitraria. L'attacco funziona su varie microarchitetture Intel prodotte sin dal 2010, indipendentemente dal sistema operativo in uso. Meltdown è quindi in grado di accedere arbitrariamente a qualsiasi locazione di memoria protetta (afferenti al kernel o ad altri processi) senza necessitare alcun permesso o privilegio da parte del sistema [10].

Meltdown rompe quindi tutti i meccanismi di sicurezza che si basano sull'isolamento degli spazi di indirizzamento, andando a colpire milioni di utenti. Il sistema di protezione KAISER, sviluppato originariamente per KASLR [2], ha l'importante effetto secondario di impedire l'utilizzo di Meltdown [10].

## 2.1 Background

### 2.1.1 Esecuzione Fuori Ordine

### 2.1.2 Spazi di indirizzamento

Per risolvere diversi problemi, in particolare l'isolamento dei processi [7], le CPU supportano l'utilizzo di spazi d'indirizzamento virtuali, in cui gli indirizzi virtuali (relativi al singolo processo) vengono tradotti in indirizzi fisici. Lo

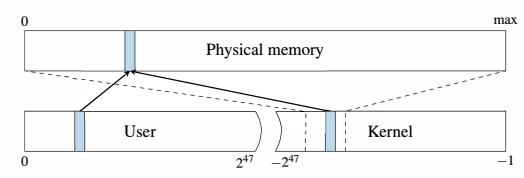


Figura 2.1: text

spazio d'indirizzamento di un processo (ovvero tutti i possibili indirizzi che un processo può generare) viene suddiviso in regioni dette *pagine* che possono essere mappate individualmente nella memoria fisica attraverso una tabella di traduzione multivello. Ogni processo possiede una propria tabella di traduzione che traduce tutti e soli i suoi indirizzi virtuali e che definisce le proprietà di protezione delle varie zone di memoria.

Ogni processo può quindi riferirsi esclusivamente agli indirizzi appartenenti al proprio spazio di indirizzamento virtuale. Per permettere l'utilizzo

### 2.1.3 Attacchi Cache

Al fine di velocizzare gli accessi alla RAM, le CPU contengono buffer di memoria molto veloce ma di dimensioni limitate che costituiscono la cosiddetta *memoria cache*. La memoria cache maschera i tempi di latenza estremamente lunghi per l'accesso alla memoria centrale (molto lenta in confronto alla cache) conservando le locazioni di memoria che, secondo principi statistici come la *località spaziale* (se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo accederà ad un indirizzo vicino) e la *località temporale* (se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo vi accederà di nuovo), è più probabile vengano indirizzate dalla CPU nel breve periodo [6].

Gli attacchi a canale laterale (*side-channel attacks*) contro la cache sfruttano questa differenza di tempo di accesso introdotta dalla cache stessa. Negli attacchi Flush+Reload [11], usati da Meltdown [10], l'attaccante è in grado di determinare se una locazione di memoria è stata precedentemente caricata in cache, misurando il tempo impiegato da un'operazione di lettura.

## 2.2 Come agisce Meltdown

L'attacco Meltdown consiste in tre passi fondamentali [10]:

1. Leggere il contenuto di una locazione di memoria inaccessibile dall'attaccante, causando il lancio di un'eccezione di protezione

2. Accedere in maniera speculativa ad una linea di memoria cache in base al contenuto segreto della locazione protetta

3. Usare un'attacco di tipo Flush+Reload per determinare il contenuto segreto in base a quale linea di memoria è stata acceduta

### 2.2.1 Passo 1: Leggere il segreto

Nel prima passo di Meltdown, l'attaccante cerca di accedere ad una zona di memoria protetta, ad esempio la memoria kernel. Il tentativo di accesso ad una pagina non accessibile da livello utente fa in modo che la CPU sollevi un'eccezione di protezione, che generalmente termina il processo. Tuttavia, a causa dell'esecuzione fuori ordine, la CPU potrebbe aver già eseguito l'istruzione di accesso in maniera speculativa *prima* delle istruzioni relative all'eccezione di protezione, al fine di minimizzare i tempi di latenza (vedi paragrafo 2.1.1). In questo modo la CPU accederebbe in maniera speculativa alla locazione di memoria desiderata prima che il processo venga terminato.

Grazie al lancio dell'eccezione, le eventuali istruzioni eseguite in maniera speculativa, che non sarebbero docute essere eseguite in quanto relative ad una previsione di salto *errata*, non vengono *ritirate* dalla CPU e non hanno così alcun effetto sulla macroarchitettura in generale (memoria centrale e registri logici non speculativi del processore) [1].

### 2.2.2 Passo 2: Trasmettere il segreto

Per poter trasmettere il segreto, si utilizza un *probe array*, di dimensione pari a 256 pagine virtuali e allocato precedentemente nella memoria del processo attaccante, assicurandosi che *nessuna porzione dell'array sia presente nella cache*. La sequenza di transient instruction contiene un accesso ad un elemento del probe array, il cui offset è calcolato moltiplicando il valore del byte per la dimensione di una pagina virtuale (tipicamente e nel nostro sistema è $4KB$ [7]).

Quando la CPU gestisce l'eccezione di protezione causata dal Meltdown, le transient instruction non vengono ritirate dalla CPU, senza avere dunque effetti a livello di macroarchitettura. Sebbene quindi non sia possibile rendere direttamente disponibile il segreto dal programma utente, si hanno importanti

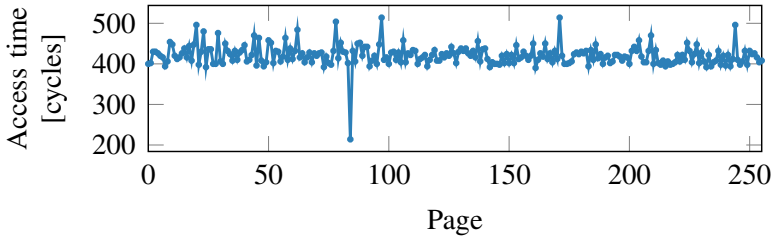


Figura 2.2: Tempo di accesso alle 256 pagine del probe array. Il grafico mostra una *cache hit* sulla pagina acceduta nel passo 2. [10]

effetti secondari a livello di microarchitettura, in particolare nella memoria cache [10].

Durante l'esecuzione speculativa, infatti, la locazione di memoria all'interno del probe array che viene acceduta dalla CPU, viene memorizzata in memoria cache e vi rimane anche in seguito all'annullamento degli effetti delle transient instruction, rendendola vulnerabile ad un attacco side-channel.

L'utilizzo del valore segreto moltiplicato per la dimensione della pagina, ci garantisce sia una precisa *correlazione* tra il valore segreto e la locazione caricata in memoria, sia che a differenti valori della locazione di memoria saranno accedute differenti *pagine* del probe array. Ciò previene il fatto che il prefetcher hardware (per ragioni di ottimizzazione) potrebbe caricare in cache anche le locazioni di memoria adiacenti a quella acceduta, rendendo impossibile determinare quale locazione di memoria sarebbe stata indirizzata se non fosse stato utilizzato a priori questo accorgimento.

### 2.2.3 Passo 3: Ricevere il segreto

Dopo che la sequenza di istruzioni del passo 2 è stata eseguita, in cache è presente esattamente una linea di memoria del probe array. L'offset di questa linea è dipendente esclusivamente dal valore segreto presente nell'arbitaria locazione di memoria protetta. Grazie a ciò, l'attaccante può effettuare un'attacco Flush+Reload [11], iterando attraverso le 256 pagine del probe array e misurando il tempo di accesso per il primo elemento di ogni pagina (vedi figura 2.2). In base a quanto detto finora, la pagina con la latenza minore è l'unica presente in memoria cache e il numero della pagina è il valore segreto letto dalla memoria protetta.

### 2.2.4 Conclusioni

Il seguente codice mostra in Assembly x86-64 la sequenza di istruzioni alla base di Meltdown, relative ai passi 1 e 2 dell'attacco.

```
1 # rcx = indirizzo di memoria kernel
2 # rbx = probe array
3 movb (%rcx), %al          # Lettura del segreto
4 shl  $12, %rax            # Traslazione dell'offset
5 movq (%rbx, %rax), %rbx   # Trasmissione del segreto
```

Meltdown è quindi in grado di leggere in maniera arbitraria dati presenti in memoria *protetta*, ad esempio nello spazio di indirizzamento kernel. L'efficienza di Meltdown si basa principalmente sull'esistente *race condition* tra il lancio dell'eccezione di protezione e il passo 2 del nostro attacco (vedi paragrafo 2.2.2 a pagina 9). Per questo motivo, sono previsti alcuni accorgimenti e ottimizzazioni ulteriori non significativi per la nostra trattazione e per le quali rimando a Lipp et al. [10].

Dato che l'intera memoria fisica viene mappata all'interno dello spazio di indirizzamento del kernel attraverso la cosiddetta *finestra di memoria fisica* [8], Meltdown è in grado non solo di leggere le zone di memoria relative al kernel, ma anche gli spazi di memoria di tutti gli altri processi. In base a quanto rilevato da Lipp et al. [10], Meltdown è in grado di effettuare il dump dell'intera memoria fisica fino ad una velocità di 503 KB/s.

## 2.3 Il sistema di protezione KAISER

KAISER, proposto da Gruss et al. [2], è una modifica del nucleo in cui il kernel non viene mappato nello spazio virtuale dei processi utente. Questa modifica era stata pensata per prevenire attacchi side-channel contro la misura di protezione KASLR [4, 3, 5], ma ha l'importante effetto secondario di prevenire Meltdown [10]

L'idea alla base di KAISER è quella di separare lo spazio di memoria kernel da quello utente, ovvero di rendere disponibile la traduzione degli indirizzi virtuali del kernel soltanto quando il sistema si trova in modalità privilegiata. Ciò previene Meltdown in quanto, quando il sistema sta eseguendo il programma utente della sezione 2.2.4, la traduzione dell'indirizzo virtuale scelto non è presente nell'albero di traduzione del processo attaccante e dunque la CPU non è in grado di accedervi nemmeno in maniera speculativa.

# Capitolo 3

# Introduzione al nucleo didattico

# Capitolo 4

# L'implementazione del sistema di protezione

# Bibliografia


[1] Graziano Frosini e Giuseppe Lettieri. *Architettura dei calcolatori Vol. II*. A cura di Pisa University Press. 2013.

[2] Daniel Gruss et al. «KASLR is Dead: Long Live KASLR». In: *International Symposium on Engineering Secure Software and Systems*. Springer International Publishing, 2017, pp. 161–176.

[3] Daniel Gruss et al. «Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR». In: *Conference on Computer and Communications Security*. 2016, pp. 368–379.

[4] Ralf Hund, Carsten Willems e Thorsten Holz. «Practical Timing Side Channel Attacks against Kernel Space ASLR». In: *Security and Privacy*. 2013, pp. 191–205.

[5] Yeongjin Jang, Sangho Lee e Taesoo Kim. «Breaking Kernel Address Space Layout Randomization with Intel TSX». In: *Conference on Computer and Communications Security*. 2016, pp. 380–392.

[6] Giuseppe Lettieri. *Memoria Cache*. 16 Mar. 2017. URL: `https://calcolatori.iet.unipi.it/resources/cache.pdf`.

[7] Giuseppe Lettieri. *Paginazione*. 3 Mag. 2019. URL: `https://calcolatori.iet.unipi.it/resources/paginazione.pdf`.

[8] Giuseppe Lettieri. *Paginazione: complementi*. 4 Apr. 2017. URL: `https://calcolatori.iet.unipi.it/resources/tlb.pdf`.

[9] Giuseppe Lettieri. *Protezione*. 11 Apr. 2019. URL: `https://calcolatori.iet.unipi.it/resources/protezione.pdf`.

[10] Moritz Lipp et al. «Meltdown: Reading Kernel Memory from User Space». In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[11] Yuval Yarom e Katrina Falkner. «FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack». In: *USENIX Security Symposium*. 2014, pp. 719–732.